La récursivité ou l'algorithmique sans boucles

Roger Cuppens(*)

Dans les programmes du secondaire, sont introduits des éléments d'algorithmique dont l'essentiel semble être l'étude des boucles de calcul. Nous montrons ici qu'il existe une méthode pour « calculer sans boucles » qui semble plus proche de l'apprentissage usuel des mathématiques.

1. Le calculable

Dans la littérature, on trouve que l'*algorithmique* est l'ensemble des règles et des techniques qui sont impliquées dans la définition et la conception d'algorithme.

Par contre, il est plus difficile de trouver une définition précise du mot algorithme. On peut proposer la définition suivante :
Un *algorithme* est une procédure de calcul automatique qui fournit en un nombre fini d'étapes le résultat escompté.

Mais qu'est-ce que *calculer* ? Ce problème a été résolu dans les années 1930 de plusieurs manières très différentes. La plus connue, semble-t-il, est la *machine de Turing*, modèle théorique dont les ordinateurs actuels sont une réalisation pratique. La description du fonctionnement de cette machine a donné naissance à la programmation traditionnelle.

Une autre approche totalement différente, *le lambda-calcul*, avait été proposée précédemment par Alonso Church. Le fait que ces deux théories (et d'autres dont je ne parlerai pas ici) sont équivalentes ont amené Church à penser ce qu'on appelle maintenant la *thèse de Church* selon laquelle ces méthodes fournissent bien les modèles du calcul.

Pour Church, calculer revient à appliquer une fonction à des données pour obtenir un résultat. Le lambda-calcul consiste en l'étude de la construction de ces fonctions qui pour certaines doivent pouvoir être *récursives*, c'est-à-dire pouvoir se définir à partir d'elles-mêmes. L'équivalence énoncée plus haut permet donc de dire qu'***un algorithme est une fonction*** et que ***l'algorithmique est l'étude des méthodes permettant d'obtenir de telles fonctions***.

C'est cette conception que nous utiliserons dans la suite : des fonctions données *a prio*ri, les *primitives*, définissent un *univers* dans lequel on peut définir de nouvelles fonctions résolvant le problème posé.

Pour bien marquer la différence entre algorithmique et programmation qui n'est pas toujours évidente dans la littérature, nous ne nous préoccupons pas dans les exemples qui suivent de savoir si les calculs vont être exécutés « à la main » ou en

(*) IRES de Toulouse. roger.cuppens@orange.fr.

utilisant une machine, les considérations relatives à l'utilisation de cette dernière étant renvoyées au dernier paragraphe.

2. Premiers exemples

Dans ce paragraphe, l'univers sera celui des entiers naturels, les primitives étant les opérations d'addition, de soustraction et de multiplication notées traditionnellement +, – et × et les relations d'ordre ≤ et ≥ et la primitive logique si … alors …. Une définition différente de l'univers des entiers naturels sera donnée dans la suite.

Nous utilisons dans la suite une syntaxe proche de celle utilisée en mathématiques. Par exemple nous écrivons

$$\text{carré}(p) = p \times p$$

le symbole « = » étant ici un symbole de définition.

2.1. Premier exemple : la factorielle

Nous commençons par la fonction factorielle que nous noterons fact, la notation traditionnelle « ! » ayant tendance à faire oublier l'aspect fonctionnel. On connait tous la définition :

$$\text{fact}(n) = 1 \times 2 \times \cdots \times (n-1) \times n.$$

L'algorithmique traditionnelle nous apprend que pour calculer les valeurs d'une telle fonction on fait appel à une boucle de programmation. Mais si on remarque que le groupement des $(n - 1)$ premiers termes nous donne la formule :

$$\text{fact}(n) = n \times \text{fact}(n-1),$$

et puisque fact(1) est évidemment égal à 1[1], on peut écrire la fonction fact sous la forme suivante :

$$\begin{aligned} \text{fact}(n) = \text{si}(n = 1) &\text{ alors } 1 \\ &\text{sinon } n \times \text{fact}(n-1) \end{aligned} \tag{2.1}$$

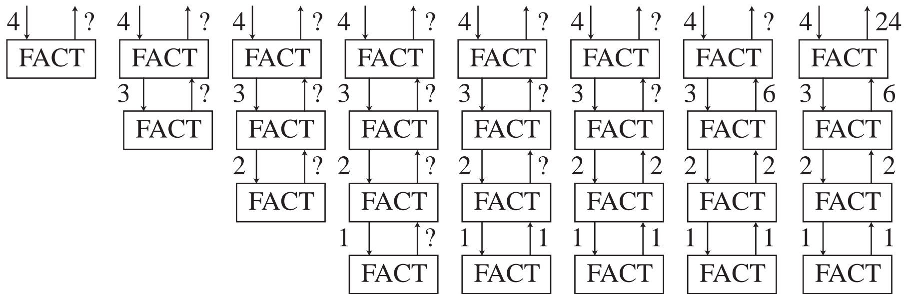
dite *récursive* car la fonction fact est alors définie à partir d'elle-même.

Pour comprendre comment on peut calculer avec cette définition, on peut considérer cette fonction comme une « boîte noire » $n\downarrow$ FACT $\uparrow ?$ où on entre une valeur « n » et où attend le résultat « ? ».

Par exemple, si on veut calculer fact(4), on partira de $4\downarrow$ FACT $\uparrow ?$ qui fera appel à $3\downarrow$ FACT $\uparrow ?$, elle-même faisant appel à $2\downarrow$ FACT $\uparrow ?$ et finalement à $1\downarrow$ FACT $\uparrow ?$. On aura

(1) On peut définir le produit d'un ensemble vide de nombres comme étant égal à 1, ce qui donne ici fact(0) = 1 ; dans ce cas il suffit dans la formule (2.1) de remplacer la condition $(n = 1)$ par $(n = 0)$.

successivement les résultats : FACT (1↓ ↑1), FACT (2↓ ↑2), FACT (3↓ ↑6) et finalement FACT (4↓ ↑24). En résumé on a les huit étapes suivantes



C'est dans ce processus de « descente » vers la valeur d'arrêt 1, puis de « remontée » des résultats que réside le principe du calcul récursif.

Remarques.

• On a un schéma général des algorithmes de calcul des suites de la forme

$$u_{n+1} = f(u_n).$$

• En algorithmique, on insiste sur la nécessité de démontrer la validité de l'algorithme fourni. Ceci revient à répondre aux deux questions suivantes :
– l'algorithme va-t-il donner une réponse en un temps fini ?
– l'algorithme va-t-il fournir la « bonne » réponse ?
On voit qu'ici il suffit d'invoquer la descente infinie (c'est-à-dire le fait qu'une suite d'entiers naturels strictement décroissante est finie) pour répondre à la première et une récurrence simple pour la deuxième.

2.2. La puissance

Un exemple analogue est la fonction

$$\text{puissance}(x,n) = \text{si}(n=0)\text{ alors }1$$
$$\text{sinon } x \times \text{puissance}(x,n-1)$$

permettant de calculer x^n qui porte sur deux variables, mais bien entendu la récurrence ne porte que sur la seconde.

2.3. La suite de Fibonacci

On sait que la suite de Fibonacci est définie par :

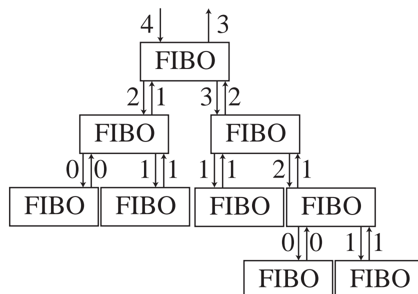
$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ F_{n+2} = F_n + F_{n+1} \end{cases}$$

On peut donc penser à la fonction suivante :

$$\text{fibo}(n) = \text{si}\,(n=0)\ \text{alors}\ 0 \quad \text{sinon}\ \text{si}\,(n=1)\ \text{alors}\ 1 \quad \text{sinon}\ \text{fibo}(n-2)+\text{fibo}(n-1) \tag{2.2}$$

Mais on peut faire deux remarques :

– si on utilise cette fonction on constate que pour simplement obtenir fibo(4) = 3, on a 9 appels de l'algorithme :



et le calcul devient rapidement fastidieux, voire irréalisable.

– elle ne correspond pas au raisonnement plus naturel et plus économique : puisque $F_0 = 0$ et $F_1 = 1$, on a immédiatement $F_2 = 1$ et puisque $F_1 = 0$ et $F_2 = 1$, on a $F_3 = 2$, et puisque $F_2 = 1$ et $F_3 = 2$, finalement $F_4 = 3$.

Nous verrons dans la suite comment obtenir un algorithme conforme à cette méthode.

2.4. Le pgcd

On sait que dans le livre VII des *Éléments* d'Euclide (vers 300 av. J.C.) figure une méthode pour calculer le pgcd de deux naturels non nuls qui est maintenant connue sous le nom d'algorithme d'Euclide et est considérée comme l'un des plus vieux algorithmes de l'histoire des mathématiques. Elle consiste à diviser le plus grand nombre par le plus petit, à diviser le plus petit par le reste de cette division et ainsi de suite jusqu'à ce que le reste obtenu soit nul : le pgcd est alors le dernier reste non nul. Mais, si $q \neq 0$, le calcul du reste d'une division peut s'obtenir à partir d'une suite de soustractions :

$$\text{reste}(p,q) = \text{si}\,(p<q)\ \text{alors}\ p\ \text{sinon}\ \text{reste}(p-q,p) \tag{2.3}$$

Il semble préférable d'utiliser le fait que

$$\text{si}(p>q)\ \text{alors}\ \text{pgcd}(p,q) = \text{pgcd}(p-q,q)$$

et

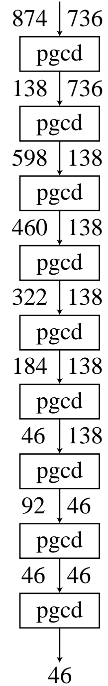
$$\text{si}(p<q)\ \text{alors}\ \text{pgcd}(p,q) = \text{pgcd}(q-p,p)$$

et puisque on a évidemment

$$\text{pgcd}(p,p) = p$$

on obtient finalement la fonction

$$\begin{aligned}\text{pgcd}(p,q) = {} & \text{si}(p = q) \text{ alors } p \\ & \text{si}(p > q) \text{ alors } \text{pgcd}(p - q, q) \\ & \text{si}(p < q) \text{ alors } \text{pgcd}(q - p, p)\end{aligned}$$



Cet exemple est intéressant à plus d'un titre :

– La variable de récurrence (p,q) appartient à $\mathbb{N}^2$ et non à $\mathbb{N}$ comme précédemment ; néanmoins l'algorithme se termine toujours puisque la suite des sommes $p + q$ est strictement décroissante.

– Contrairement aux exemples précédents où l'algorithme était une simple réécriture de la définition mathématique, la fonction s'obtient en partitionnant l'ensemble $\mathbb{N}^2$ en trois ensembles et en utilisant une propriété différente pour chacun des ensembles de la partition.

– Comme le montre l'exemple ci-dessus, quand on arrive à la condition d'arrêt $p = q$ le calcul est terminé et il n'y a donc pas de raison de faire remonter cette valeur : on parle dans ce cas de ***récursivité terminale***(2).

3. Les listes

3.1. L'univers des listes

Une *liste* est une suite finie d'éléments que nous noterons avec des crochets : $[x_1, x_2, \dots, x_n]$ sera une liste composée de n éléments x_j qui peuvent être des objets simples ou des listes. Par exemple, voyelles = $[a, e, i, o, u, y]$ est une liste de six lettres, [] est une liste n'ayant pas d'élément (*liste vide*) et [[]] une liste à un élément.

Pour manipuler ces listes, outre la liste vide, on utilisera quatre primitives :

– prem (l) fournit le premier élément d'une liste l non vide ;

– mp (x,l)(3) ajoute à la liste l l'élément x comme premier élément ;

– sp (l) enlève le premier élément d'une liste non vide l ;

– le prédicat(4) vide? (l) qui donne vrai ou faux suivant que la liste l est vide ou non.

On peut alors définir d'autres fonctions utiles, par exemple :

– une fonction concat (l, m) qui fournit la concaténation de deux listes l et m :

$$\begin{aligned}\text{concat}(l,m) = {} & \text{si vide?}(l) \text{ alors } m \\ & \text{sinon mp}\big(\text{prem}(l), \text{concat}(\text{sp}(l), m)\big)\end{aligned}$$

(2) On peut faire la même remarque pour la formule (2.3).
(3) mp est l'abréviation de metspremier tandis que sp est celle de saufpremier.
(4) Un prédicat est une fonction qui prend deux valeurs, par exemple « vrai » ou « faux ».

– une fonction longueur (l) qui fournit le nombre d'éléments d'une liste l :

$$\text{longueur}(l) = \text{si vide?}(l) \text{ alors } 0$$
$$\text{sinon } 1 + \text{longueur}(\text{sp}(l))$$

– un prédicat appartient? (x,l) qui fournit la valeur « vrai » si x est un élément de la liste l et « faux » dans le cas contraire :

$$\text{appartient?}(x,l) = \text{si vide?}(l)$$
$$\text{alors faux}$$
$$\text{sinon si}(x = \text{prem}(l))$$
$$\text{alors vrai}$$
$$\text{sinon appartient?}(x,\text{sp}(l))$$

Remarques.

• La récursivité sur les listes est légèrement différente de celle du paragraphe précédent car l'ensemble des listes n'est que partiellement ordonné pour la relation d'inclusion. Néanmoins la notion de longueur permet de ramener immédiatement le problème de la terminaison de ces algorithmes à la descente infinie sur les entiers.

• Avec ces fonctions, on peut immédiatement définir et manipuler des *piles* où le dernier arrivé est le premier servi. Par contre la manipulation des *files d'attente* où le premier arrivé doit être le premier servi nécessite d'autres fonctions, par exemple der(l) fournissant le dernier élément d'une liste non vide dont nous laissons la définition au lecteur.

• Lorsqu'une liste a deux éléments, l'on parlera du couple de ces deux éléments. Dans ce cas on peut définir deux fonctions utiles :

$$\text{second}(c) = \text{prem}(\text{sp}(c))$$

$$\text{couple}(p,q) = \text{mp}(p,\text{mp}(q,[\ ]))$$

3.2. Retour sur les factorielles

On obtient facilement que l'algorithme

$$\text{listefact}(n) = \text{si}(n=1) \text{ alors } [1]$$
$$\text{sinon mp}(n \times \text{prem}(\text{listefact}(n-1)),\text{listefact}(n-1))$$

peut fournir la suite des n premières factorielles en ordre décroissant. Mais, en raison de la double présence de listefact(n – 1) dans la définition, on risque de retomber sur le problème des calculs inutiles signalé plus haut. Pour éviter cela, il suffit de transférer le calcul à une fonction auxiliaire, ce qui donne :

$$\text{listefact}(n) = \text{si}(n=1) \text{ alors } [1] \quad \text{sinon factaux}(n, \text{listefact}(n-1)) \tag{3.1}$$
$$\text{factaux}(n,l) = \text{mp}(n \times \text{prem}(l), l)$$

3.3. Retour sur la suite de Fibonacci

Les idées précédentes fournissent immédiatement l'algorithme suivant pour la suite de Fibonacci :

$$\text{listefibo}(\text{n}) = \text{si}(\text{n}=1) \text{ alors } [1\ \ 0] \quad \text{sinon fibaux}(\text{listefibo}(\text{n-}1))$$
$$\text{fibaux}(l) = \text{mp}((\text{prem}(l) + \text{prem}(\text{sp}(l))), l)$$

Mais il est plus simple d'utiliser des couples, ce qui donne

$$\text{fibo}(n) = \text{prem}(\text{couplefibo}(n))$$
$$\text{couplefibo}(n) = \text{si}(n=1) \text{ alors } [1,0] \quad \text{sinon couplefibaux}(\text{couplefibo}(n-1)) \tag{3.2}$$
$$\text{couplefibaux}(l) = \text{couple}(\text{prem}(l) + \text{second}(l), \text{prem}(l))$$

Remarque. La méthode précédente permet de calculer toutes les suites de la forme

$$u_{n+1} = f(u_n, u_{n-1}).$$

4. La récursivité en arithmétique

4.1. L'arithmétique de Peano

Comme l'a rappelé Pierre Legrand [5] dans son remarquable article sur la récurrence, à la fin du dix-neuvième siècle, Peano a fourni une axiomatique d'un ensemble $\mathbb{N}$ à partir d'un nombre 0 et d'une fonction successeur que nous notons ici succ (l'inverse étant la fonction pred) et, à la suite de Hilbert, la plupart des mathématiciens ont cru et certains croient encore que cet ensemble est l'ensemble des entiers naturels. Or depuis Gödel on sait que l'ensemble $\mathbb{N}$ de Peano n'a pas toutes les qualités espérées : certes l'ensemble des entiers naturels peut être considéré comme un modèle de $\mathbb{N}$, mais l'existence de modèles non dénombrables de $\mathbb{N}$ (l'article de Michèle Artigue & Ferdinando Arzarello [1] en fournit un exemple) montre qu'une propriété essentielle manque dans l'axiomatique de Peano, à savoir la descente infinie dont une des formulations est : si n est un entier naturel, l'ensemble $[[1,n]]$ des entiers compris entre 1 et n est fini. Mais comme la définition intuitive d'un ensemble fini E est justement l'existence d'une bijection entre E et un des intervalles $[[1,n]]$, on a un magnifique cercle vicieux. C'était bien ce qu'avaient

pressenti les intuitionnistes quand ils prétendaient que l'existence des entiers naturels ne pouvait reposer que sur une intuition commune. On voit que le formalisme de Hilbert et à sa suite le bourbakisme s'éloignent de plus en plus de cette intuition. Faut-il y croire ? Pas forcément, comme je l'ai rappelé dans [3] et de toute façon ce n'est pas un problème à aborder au lycée et même après…

Comme un algorithme ne doit comporter qu'un nombre fini d'étapes, l'ensemble des entiers considéré doit comporter la propriété de descente infinie.

4.2. Les opérations en arithmétique de Peano

On part d'un nombre 0 et on définit les nombres entiers à partir de ce 0 avec une fonction successeur que nous notons ici « succ » dont l'inverse définie pour tout entier différent de 0 sera notée « pred ». Nous allons montrer dans ce paragraphe comment on peut calculer avec quatre symboles : les trois qui précèdent et l'égalité « = ».

Pour obtenir la somme de deux nombres p et q, puisque, si $q = 0$, $p + q = p$ et $p + q = (1 + p) + (q - 1)$ sinon, on a immédiatement :

$$\begin{aligned}\text{somme}(p,q) = \text{si}(q=0)\ &\text{alors } p\\ &\text{sinon succ}\left(\text{somme}\left(p,\text{pred}(q)\right)\right)\end{aligned}$$

et pour obtenir le produit de deux nombres p et q, puisque $0 \times q = 0$ et si $p > 0$, $p \times q = (p - 1) \times q + q$ on a :

$$\begin{aligned}\text{produit}(p,q) = \text{si}(p=0)\ &\text{alors } 0\\ &\text{sinon somme}\left(\text{produit}\left(\text{pred}(p),q\right),q\right)\end{aligned}$$

Pour obtenir la différence de deux nombres p et q, puisque $p - 0 = p$ et si $q > 0$, alors $0 - q$ est impossible (ce que nous notons $0 - q = \text{nil}$) et si p est aussi positif, alors $p - q = (p - 1) - (q - 1)$ on a :

$$\begin{aligned}\text{diff}(p,q) = \text{si}(q=0)\ &\text{alors } p\\ &\text{sinon si}(p=0) \text{ alors nil}\\ &\qquad \text{sinon diff}\left(\text{pred}(p),\text{pred}(q)\right)\end{aligned}$$

Pour la division, nous aurons besoin d'un prédicat que nous noterons en notation fonctionnelle inf?, inf? (p, q) étant ce que l'on note traditionnellement $p < q$. Si on utilise les remarques suivantes :

– $p < 0$ est faux,

– si $q > 0$, $0 < q$ est vrai,

– si $p > 0$ et $q > 0$, $p < q \Leftrightarrow p - 1 < q - 1$,

on a immédiatement la fonction :

$$\text{inf?}(p,q) = \text{si}\,(q=0)\ \text{alors}\ \ \text{faux}$$
$$\text{sinon}\ \ \text{si}\,(p=0)\ \text{alors}\ \ \text{vrai}$$
$$\text{sinon}\ \ \text{inf?}\big(\text{pred}(p),\text{pred}(q)\big)$$

On a alors facilement pour le quotient et le reste de p par q :

$$\text{quot}(p,q) = \text{si}\,(q=0)\ \text{alors}\ \ \text{nil}$$
$$\text{sinon}\ \ \text{si}\,\big(\text{inf?}(p,q)\big)\ \text{alors}\ \ 0$$
$$\text{sinon}\ \ \text{succ}\big(\text{quot}\big(\text{diff}(p,q),q\big)\big)$$

$$\text{reste}(p,q) = \text{si}\,(q=0)\ \text{alors}\ \ \text{nil}$$
$$\text{sinon}\ \ \text{si}\,\big(\text{inf?}(p,q)\big)\ \text{alors}\ \ p$$
$$\text{sinon}\ \ \text{reste}\big(\text{diff}(p,q),q\big)$$

Avec la notion de couple, on peut même définir la division de p par q comme une fonction qui associe à p et q le couple composé du quotient et du reste par :

$$\text{div}(p,q) = \text{si}\,(q=0)$$
$$\text{alors nil}$$
$$\text{sinon si}\,\big(\text{inf?}(p,q)\big)$$
$$\text{alors couple}\ (0,p)$$
$$\text{sinon sommecouples}\big(\text{couple}\ (1,0),\big(\text{div}\big(\text{diff}(p,q),q\big)\big)\big)$$

où sommecouples désigne l'addition de deux couples :

$$\text{sommecouples}\big([x_1\ \ y_1],[x_2\ \ y_2]\big) = \text{couple}\big(\text{somme}(x_1,x_2),\text{somme}(y_1,y_2)\big)$$

4.3. Un modèle simple de ℕ

Il s'agit d'une reprise du plus vieux mode de numération connu, que nous appellerons « système buchettes » qui consiste à utiliser des tas d'un même objet, système qui est vite impraticable pour un être humain, mais que l'on peut très bien réaliser avec un ordinateur.

En effet prenons comme 0 la liste vide et comme définition du successeur d'un entier n :

$$\text{succ}(n) = \text{mp}(1,n).$$

Dans cet univers, le nombre que nous notons 8 sera donc représenté par

[1 1 1 1 1 1 1 1].

tandis que la fonction longueur fournit la représentation usuelle du nombre et la fonction concat fournit la somme de deux nombres.

5. Quelques algorithmes irrationnels

5.1. Calcul des décimales de e

Pour calculer des décimales de e, on peut partir de l'encadrement :

$$\sum_{k=0}^{n} \frac{1}{k!} < \mathrm{e} < \sum_{k=0}^{n} \frac{1}{k!} + \frac{1}{n!}$$

En posant $\sum_{k=0}^{n} \frac{1}{k!} = \frac{c_n}{n!}$ on obtient facilement que

$$\begin{cases} c_0 = 1 \\ c_n = n \times c_{n-1} + 1 \end{cases}$$

L'algorithme

$$\begin{aligned} \mathrm{c}(n) = \mathrm{si}\,(n=0)\ &\text{alors } 1 \\ &\text{sinon } n \times \mathrm{c}(n-1)+1 \end{aligned} \tag{5.1}$$

fournit donc les valeurs de c_n. Il suffit alors de diviser c_n et $c_n + 1$ par fact(n) et de conserver les décimales identiques dans les deux nombres obtenus.

5.2. La méthode d'Archimède pour calculer des décimales de π

Soit (S_n) et (T_n) deux suites de polygones réguliers respectivement inscrits et circonscrits à un cercle C de diamètre 1 définis de la manière suivante : S_1 et T_1 sont des carrés et on passe de S_n à S_{n+1} et de T_n à T_{n+1} en doublant le nombre de côtés. Si s_n et t_n sont les périmètres des polygones S_n et T_n, on peut démontrer les relations

$$t_{n+1} = \frac{2 s_n t_n}{s_n + t_n}$$

$$s_{n+1} = \sqrt{s_n t_{n+1}}$$

et on peut montrer que les suites (s_n) et (t_n) sont deux suites adjacentes de limite commune π et puisque

$$s_1 = 2\sqrt{2}, \quad t_1 = 4$$

on peut calculer le couple des nombres s_n et t_n avec l'algorithme suivant :

$$\begin{gathered} \text{archi}(n) = \text{si } n = 1 \text{ alors } \left[2\sqrt{2}, 4\right] \text{ sinon calcul}(\text{archi}(n-1)) \\ \text{calcul}(c) = \text{aux}\,1(\text{prem}(c, \text{der}(c))) \\ \text{aux}\,1(s,t) = \text{aux}\,2\left(s, \frac{2st}{s+t}\right) \\ \text{aux}\,2(s,u) = \left[\sqrt{su}, u\right] \end{gathered} \tag{5.2}$$

et de nouveau les décimales identiques dans les deux nombres obtenus sont des décimales du nombre π.

5.3. La méthode de Newton

Soit C la courbe représentative d'une fonction dérivable f dans un repère orthonormé (O,x,y). Si la courbe C coupe l'axe Ox en un point A d'abscisse a et si p est un nombre proche de a, alors la tangente au point $(p,f(p))$ coupe l'axe Ox en un point P′ d'abscisse

$$p' = p - \frac{f(p)}{f'(p)}$$

et si f est une « bonne » fonction, le nombre p' est compris entre a et p. En itérant, on peut espérer que la suite

$$\begin{cases} p_0 = p \\ p_{n+1} = p_n - \dfrac{f(p_n)}{f'(p_n)} \end{cases} \tag{6.1}$$

converge vers a.

On pourrait penser à transformer ceci en

$$\begin{gathered} \text{newton}(p,n) = \text{si } n = 0 \text{ alors } p \text{ sinon newton}(\text{calcul}(p), n-1) \\ \text{calcul}(x) = x - f(x) / f'(x) \end{gathered} \tag{6.2}$$

mais ceci ne peut être considéré comme un algorithme pour plusieurs raisons :
– Où placer la fonction f : comme variable ? Ceci est possible, mais changerait le point de vue sur cette notion de variable.
– Même si c'était le cas, on n'a pas de méthode générale pour calculer la dérivée d'une fonction sauf dans un système de calcul formel et encore…
– Si ces deux problèmes étaient résolus, il n'est pas sûr qu'un tel problème donne une solution et encore moins la bonne…

On parlera donc de « schéma d'algorithme » pour cette méthode et on peut montrer que sous des conditions très fortes, par exemple quand f est analytique, la suite (p_n) converge vers a et même très rapidement !

Exemple 1. Pour la fonction $f(x) = \cos(x) - x$, la formule (6.2) devient

$$\text{calcul}(x) = x + (\cos(x) - x)/(\sin(x)+1) \tag{6.3}$$

avec laquelle on peut obtenir une valeur approchée de la racine de l'équation

$$\cos(x) = x.$$

Exemple 2. Pour la fonction $f(x) = \tan(x) - 1$, la formule (6.2) devient

$$\text{calcul}(x) = x + \cos(x)(\cos(x) - \sin(x)) \tag{6.4}$$

Plus généralement, si une fonction f a une inverse f^{-1} sur un intervalle I et si a est une valeur de $f(\mathrm{I})$, alors la méthode de Newton appliquée à $f - a$ permet de calculer des décimales de $f^{-1}(a)$.

5.4. Racine carrée

Soit a un nombre positif. Pour la fonction $f(x) = x^2 - a$, la formule (6.1) devient

$$p_{n+1} = p_n - \frac{p_n^2 - a}{2p_n} = \frac{1}{2}\left(p_n + \frac{a}{p_n}\right)$$

qui est une suite connue sous le nom de suite de Heron. Pour calculer des valeurs approchées de $\sqrt{a}$, on peut donc utiliser l'algorithme

$$\text{heron}(a,p,n) = \text{si } n = 0 \quad \text{alors } p \quad \text{sinon heron}(a,\text{calcul}(a,p),n-1)$$
$$\text{calcul}(a,x) = \frac{1}{2}\left(x + \frac{a}{x}\right) \tag{6.5}$$

Plus généralement considérons la fonction $f(x) = x^q - a$, où q est un entier positif et a un nombre positif. Comme précédemment, on obtient l'algorithme

$$\text{qracine}(q,a,p,n) = \text{si } n = 0 \text{ alors } p$$
$$\text{sinon qracine}(q,a,\text{calcul}(q,a,p),n-1) \tag{6.6}$$
$$\text{calcul}(q,a,x) = \frac{1}{q}\left((q-1)x + \frac{a}{x^{q-1}}\right)$$

fournissant des valeur approchées de $\sqrt[q]{a}$.

6. Les calculs avec une machine

Les idées de Church ont donné naissance à une classe de langages dits *langages fonctionnels* dont le prototype est le langage LISP (acronyme de LIST Programming), dont on a pu critiquer la syntaxe en disant que c'était a *Lot of Insipide and Stupid Parentheses* et la gourmandise en mémoire et en temps de calcul. Avec les ordinateurs actuels(5), ces critiques ne sont plus d'actualité.

Pour un débutant, nous luis préférons le langage LOGO conçu au départ pour initier de jeunes enfants à l'algorithmique, mais qui s'avère un langage à part entière. La syntaxe en est simple, très proche de celle que j'ai utilisée précédemment. Il en existe des versions françaises et libres. On verra aussi qu'il permet facilement des calculs sur des grands nombres, ce qui est essentiel pour cette étude.

Mais il est évident que les calculs qui suivent peuvent être réalisés avec n'importe quel langage gérant les piles de la récursivité(6).

(5) J'ai commencé avec un micro qui avait 64 k de mémoire centrale !
(6) Sinon changez-en !

6.1. La factorielle

La formule (2.1) permet de calculer « à la main » fact(n) pour des petites valeurs de la variable n. Si on utilise un langage fonctionnel, on constate que l'algorithme cherché est une simple traduction de (2.1). Par exemple dans xlogo, on écrira

```
pour fact :n
   si :n=1 [ret 1] [ret produit :n fact :n-1]
fin
```

où il est facile de comprendre que « pour … fin » est la procédure permettant de définir une nouvelle fonction dont les variables ne sont pas parenthésées, « :n » désigne la variable n et « ret » est la primitive indiquant à la machine la valeur à retourner en sortie.

Une fois cette fonction fact apprise par la machine, la requête ec fact 10 (« ec » est la primitive indiquant à la machine d'écrire le résultat) donne 3628800, ce que l'on peut vérifier à la main, alors que la requête ec fact 25 donne

15511210043330990000000000,

résultat manifestement inexact car fact(25) n'est pas divisible par 10^{10}, mais seulement par 10^6.

Remarque. Lorsqu'on utilise un outil de calcul, il est essentiel de vérifier la vraisemblance du résultat obtenu et, dans le cas d'un résultat apparemment aberrant, d'essayer d'en comprendre les raisons.

Avec l'algorithme (3.1), on obtient ainsi avec xlogo la liste suivante pour listefact (25) :

15511210043330990000000000 620448401733239500000000 25852016738884980000000 1124000727777608000000 51090942171709440000 2432902008176640000 121645100408832000 6402373705728000 355687428096000 20922789888000 1307674368000 87178291200 6227020800 479001600 39916800 3628800 362880 40320 5040 720 120 24 6 2 1

On peut vérifier que les 21 dernières valeurs sont correctes, mais que les quatre premières ne le sont pas. Or les cinq premières valeurs sont composées d'un nombre de 16 chiffres suivi de zéros : 4 pour fact(21), ce qui est correct, mais que les autres ont des zéros « en trop » : deux pour fact(22), trois pour fact(23), fact(24) et fact(25). On peut donc penser que la machine remplace un entier n par le nombre $r \times 10^q$, q et r étant le quotient et le reste du nombre n par 10^{17} et fait les calculs sur ces nombres. Ceci est renforcé par le fait que la notice du logiciel signale que 16 est le « nombre de décimales » avec lequel il calcule et qu'une primitive fixedecimales permet de modifier ce nombre.

En fixant ce nombre à 30, on obtient effectivement pour listefact (25) la liste :

15511210043330985984000000 620448401733239439360000 25852016738884976640000 1124000727777607680000 51090942171709440000 2432902008176640000 121645100408832000 6402373705728000

355687428096000 20922789888000 1307674368000 87178291200 6227020800
479001600 39916800 3628800 362880 40320 5040 720 120 24 6 2 1
dont on peut vérifier l'exactitude(7).

6.2. Suite de Fibonacci

Avec mon xlogo, j'ai pu obtenir fibo (20) = 6765, mais on peut difficilement aller plus loin avec la fonction (2.2).

Avec l'algorithme (3.2) et en calculant avec 250 décimales j'ai pu obtenir pour fibo(1000) :
434665576869374564356885276750406258025646605173717804024817290895365554179490518904038798400792551692959225930803226347752096896232398733224711616429964409065331879382989696499285160037044761377951668
49228875

6.3. Le nombre e

Avec le méthode du paragraphe 5.1, on obtient c(10) = 9864101. La machine nous donne alors

c(10)/10! = 2.7182818011463845,

(c(10)+ 1)/10! = 2.718281828459045,

ce qui donne

2.71828180 < e < 2.71828183.

En fixant à 200 le nombre de décimales, on obtient :

c(100) =
253686955560127297415270748212280220445147578566298142232775185987449253908386446518940485425152049793267407732328003493609513499849694176709764490323163992001
fact(100) =
933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000000,

et puisque

c(100)/fact(100) =
2.7182818284590452353602874713526624977572470936999595749669676277240766303535475945713821785251664274274663919320030599218174135966290435729003342952605956307380251882050351967424723324653614466387707

(1+c(100))/fact(100)
=2.7182818284590452353602874713526624977572470936999595749669676277240766303535475945713821785251664274274663919320030599218174135966290435729003342952605956307487402910862898659743078000605533618588861

(7) Si xlogo fournit pour le calcul d'un grand nombre entier une valeur c, pour vérifier que le nombre de décimales utilisé convient, il suffit de vérifier que 1 + c - c est bien égal à 1.

on peut donc penser que

e =
2.718281828459045235360287471352662497757247093699959574966967627724
076630353547594571382178525166427427466391932003059921817413596629
0435729003342952605956307…
(157 décimales !).

6.4. L'algorithme d'Archimède

Avec la méthode du paragraphe 5.2, on obtient :

archi (5) = [3.140331156954752 3.1441183852459034]

archi (10) = [3.141591421511201 3.1415951177495898]

archi (20) = [3.1415926535886203 3.141592653592145]

archi (30) = [3.141592653589794 3.1415926535897944]

On voit que cette méthode est assez lente et que dans la dernière valeur les dernières décimales 4 ou 44 sont incorrectes car on sait que

$$\pi = 3.141592653589793\ldots$$

On peut penser à des erreurs de calcul. D'ailleurs il suffit de calculer avec 20 décimales pour obtenir

archi (30) = [3.14159265358979323 68 3.14159265358979324 02]

6.5. La méthode de Newton

Avec (6.3) on obtient

newton (1,5) = cos (newton (1,5)) = 0.7390851332151607…

tandis que (6.4) donne

newton (5) = 0.7853981633974482

que l'on comparera à ce que donne xlogo pour π/4 :

pi/4 = 0.7853981633974483

Mais il faut bien se garder de donner une valeur réelle à ce calcul à (contrairement à la méthode d'Archimède) car elle repose sur la connaissance des fonctions circulaires…

6.6. L'algorithme de Heron

Avec (6.5), on obtient respectivement pour $\sqrt{2}$, $\sqrt{3}$, $\sqrt{5}$:

heron (2,1,5) = 1.414213562373095

heron (3,1,5) = 1.7320508075688774

heron (5,1,6) = 2.23606797749979

et avec (6.6)

qracine (3,2,1,6) = 1.2599210498948732

qui est bien une valeur approchée de $\sqrt[3]{2}$

Conclusion

Il est évident que les pages qui précèdent ne font qu'effleurer un immense sujet. Par exemple, l'étude de la suite de Fibonacci a montré que des algorithmes simples peuvent mener à des calculs longs, voire impossibles à réaliser dans la pratique. Ceci a donné naissance à la notion de complexité d'un algorithme qu'il nous est impossible d'étudier ici.

Un autre point que nous ne pouvons aborder est la non existence d'un algorithme pour des problèmes qui peuvent sembler a priori simples. Par exemple on peut montrer qu'il n'existe pas d'algorithme permettant de déterminer si un nombre est nul ou non nul (et donc de conclure si deux nombres obtenus de deux manières différentes sont égaux).

Néanmoins on peut voir que la plupart des calculs liés aux notions mathématiques du secondaire peuvent être traités avec cette méthode et surtout que la recherche des algorithmes y afférents donne un nouvel éclairage de ces notions. On en trouvera d'autres exemples dans [2].

Enfin, rappelons que les idées de Church ne sont plus restreintes à la logique et au calcul mathématique, mais s'appliquent à de nombreux autres domaines. Nous montrerons par exemple dans un prochain article [4] l'intérêt énorme de la récursivité en géométrie.

Bibliographie

[1] Michèle Artigue & Ferdinando Arzarello. *Les suites de Goodstein ou la puissance du détour par l'infini*. Bulletin de l'APMEP n° 498, p. 196-202.

[2] Roger Cuppens. *Apports de l'informatique en arithmétique*. Brochure de l'IREM de Toulouse n° 107, 1986.

[3] Roger Cuppens. *Remarques sur les suites de Goodstein*. Bulletin de l'APMEP n° 502 p. 89-95.

[4] Roger Cuppens. *La récursivité de la tortue*. À paraître

[5] Pierre Legrand. *La récurrence au fil des siècles*. Bulletin de l'APMEP n° 506 (novembre-décembre 2013) p. 600-610.